Using the Android* Native Development Kit (NDK)

Xavier Hallade, Technical Marketing Engineer

@ph0b - ph0b.com

Agenda

- The Android* Native Development Kit (NDK)
- Developing an application that uses the NDK
- Supporting several CPU architectures
- Debug and Optimization
- Some more tips
- Q&A

Android* Native Development Kit (NDK)

What is it?

- Build scripts/toolkit to incorporate native code in Android* apps via the Java Native Interface (JNI)

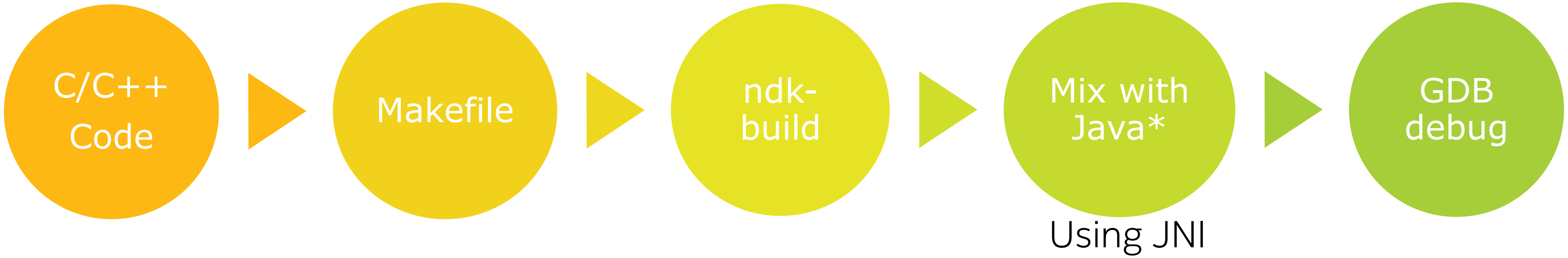
Why use it?

- Performance
  - e.g., complex algorithms, multimedia applications, games
- Differentiation
  - app that takes advantage of direct CPU/HW access
  - e.g., using SSSE3 for optimization
- Fluid and lag-free animations
- Software code reuse

Why not use it?

- Performance improvement isn't always guaranteed, in contrary to the added complexity

NDK Application Development



C/C++ Code

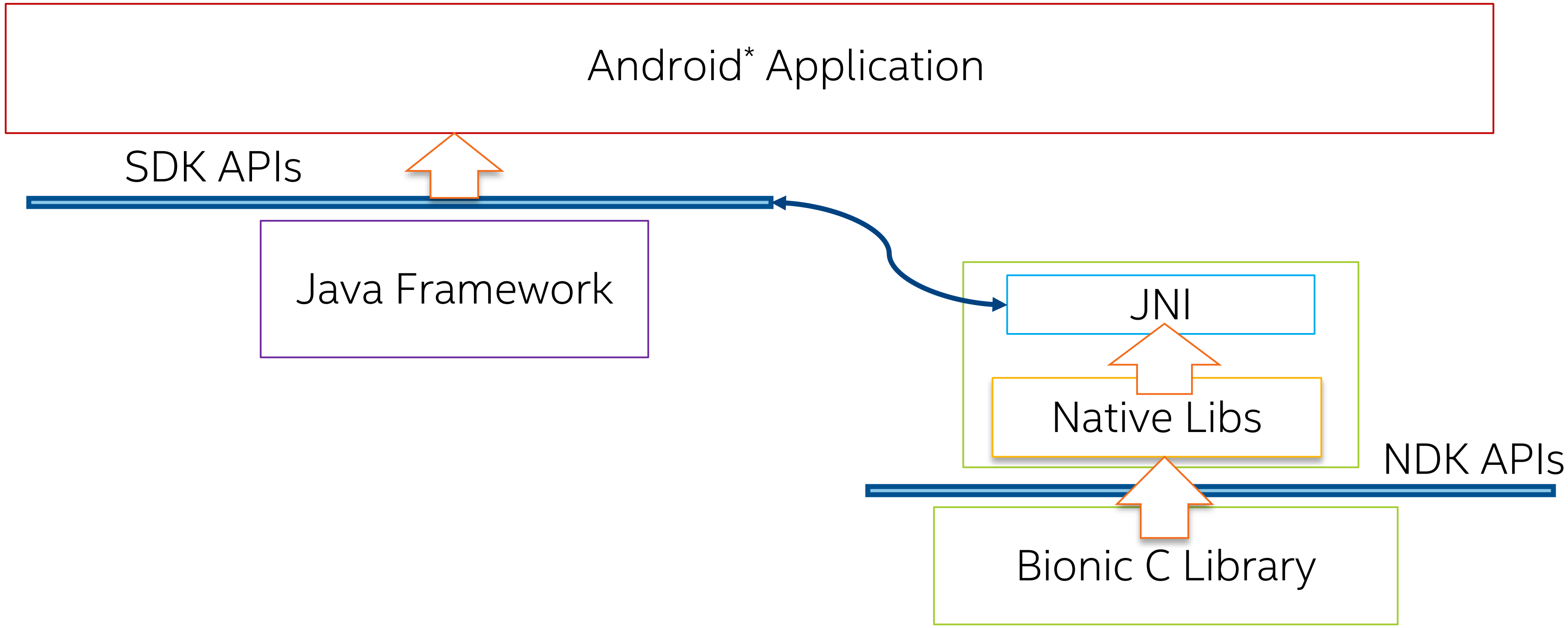
Makefile

ndk-build

Mix with Java*

Using JNI

GDB debug



Android* Application

SDK APIs

Java Framework

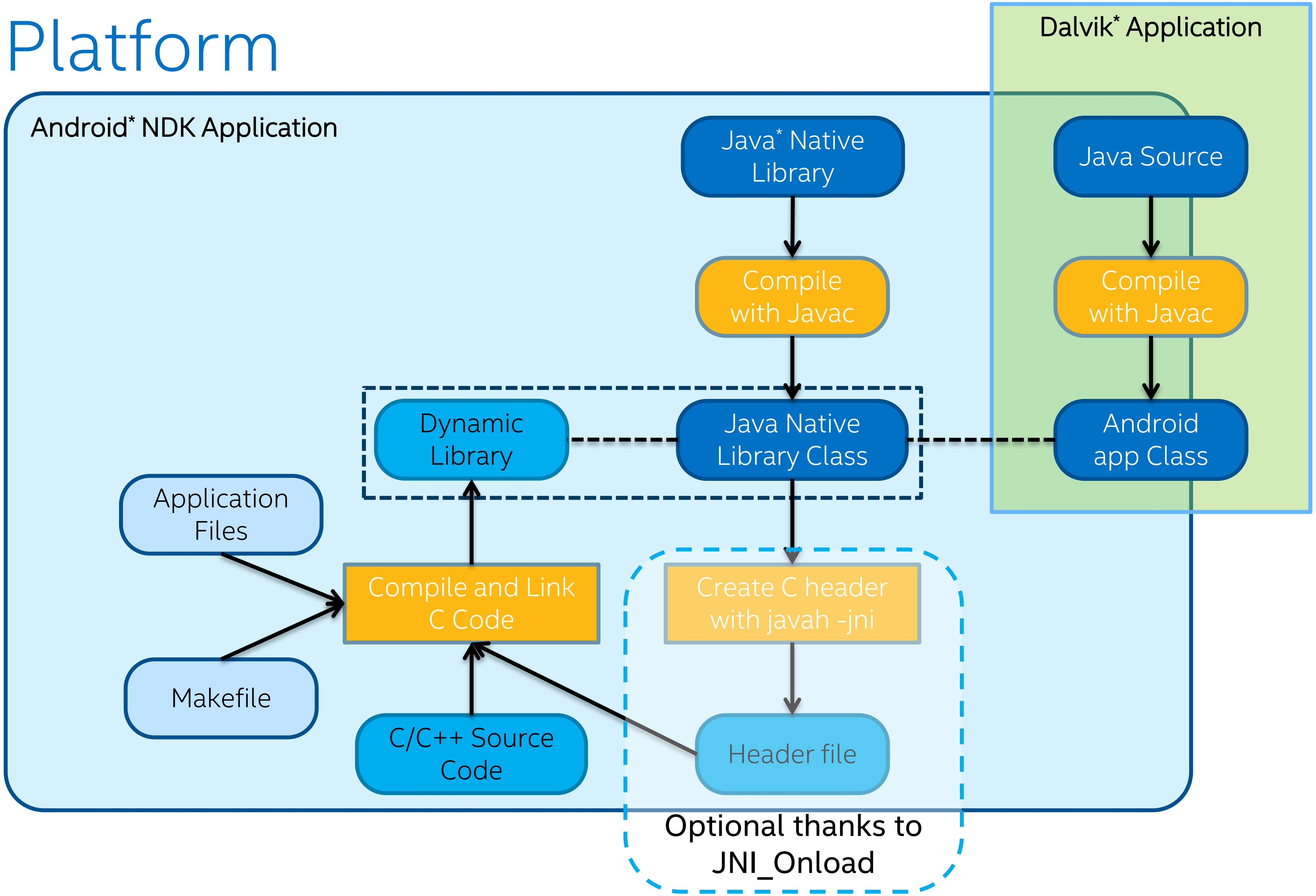
JNI

Native Libs

NDK APIs

Bionic C Library

NDK Platform



Compatibility with Standard C/C++

Bionic C Library:

- Lighter than standard GNU C Library
- Not POSIX compliant
- pthread support included, but limited
- No System-V IPCs
- Access to Android* system properties

Bionic is not binary-compatible with the standard C library

It means you generally need to (re)compile everything using the Android NDK toolchain

Android* C++ Support

By default, system is used. It lacks:

- Standard C++ Library support (except some headers)
- C++ exceptions support
- RTTI support

Fortunately, you have other libs available with the NDK:

Runtime	Exceptions	RTTI	STL
system	No	No	No
gabi++	Yes	Yes	No
stlport	Yes	Yes	Yes
gnustl	Yes	Yes	Yes
libc++	Yes	Yes	Yes

Choose which library to compile against in your Makefile (Application.mk file):

```
APP_STL := gnustl_shared
```

Postfix the runtime with _static or _shared

For using C++ features, you also need to enable these in your Makefile:

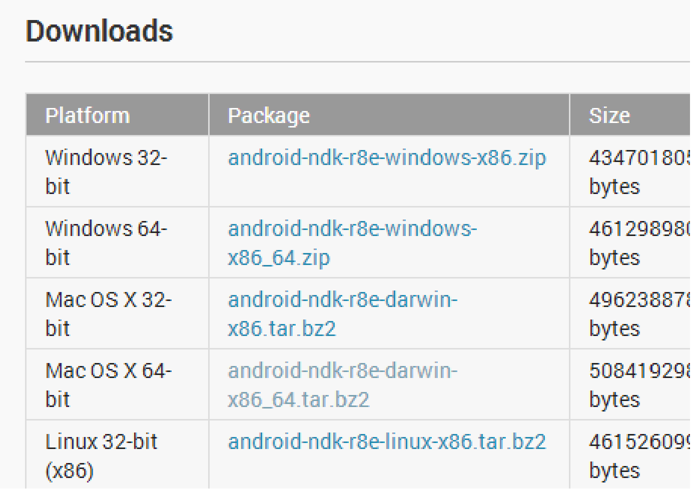
```
LOCAL_CPP_FEATURES += exceptions rtti
```

Installing the Android* NDK

NDK is a platform dependent archive:

It provides:

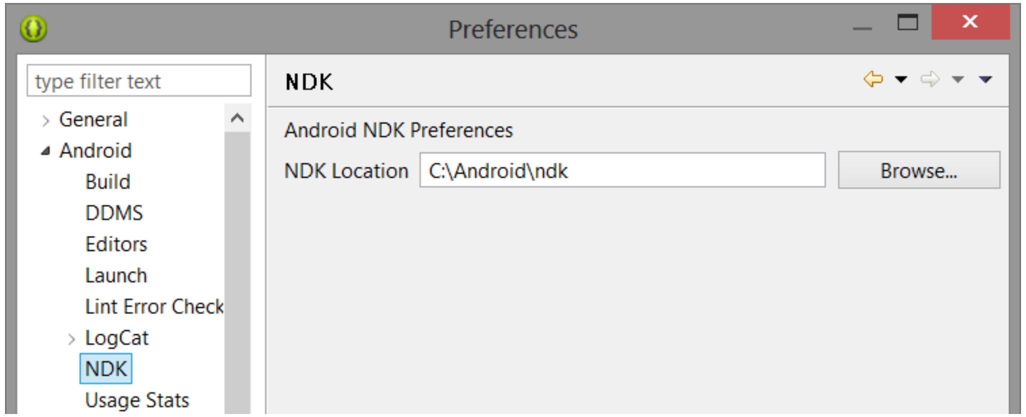
- A build environment
- Android* headers and libraries
- Documentation and samples (these are very useful)



Downloads

Platform	Package	Size
Windows 32-bit	android-ndk-r8e-windows-x86.zip	434701805 bytes
Windows 64-bit	android-ndk-r8e-windows-x86_64.zip	461298980 bytes
Mac OS X 32-bit	android-ndk-r8e-darwin-x86.tar.bz2	496238878 bytes
Mac OS X 64-bit	android-ndk-r8e-darwin-x86_64.tar.bz2	508419298 bytes
Linux 32-bit (x86)	android-ndk-r8e-linux-x86.tar.bz2	461526099 bytes

You can integrate it with Eclipse ADT:

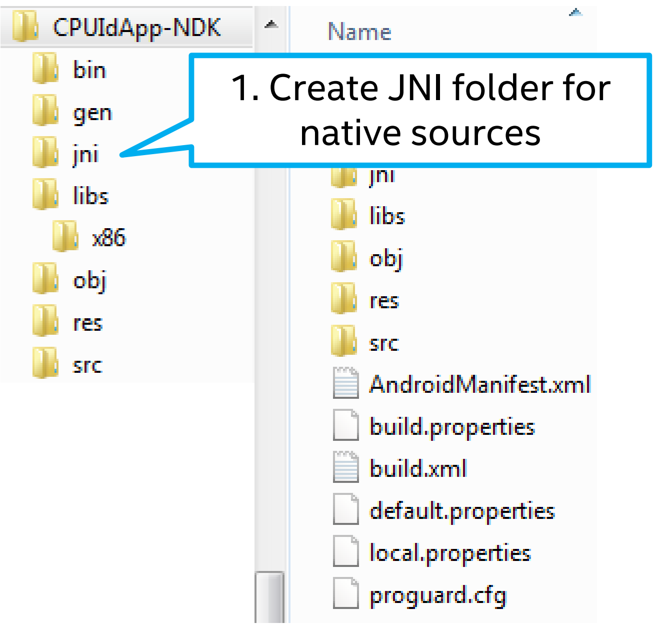


Android NDK Preferences

NDK Location C:\Android\ndk Browse...

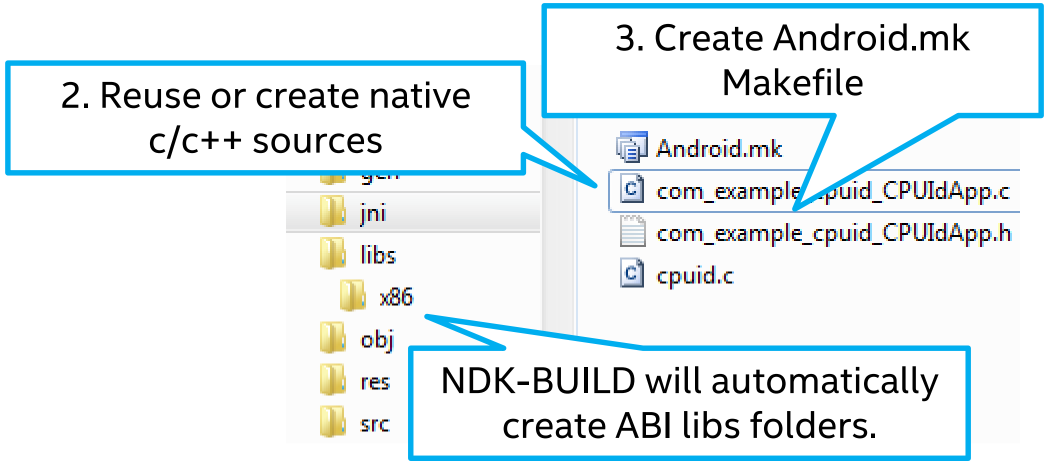
Manually Adding Native Code to an Android* Project

Standard Android* Project Structure



1. Create JNI folder for native sources

Native Sources - JNI Folder



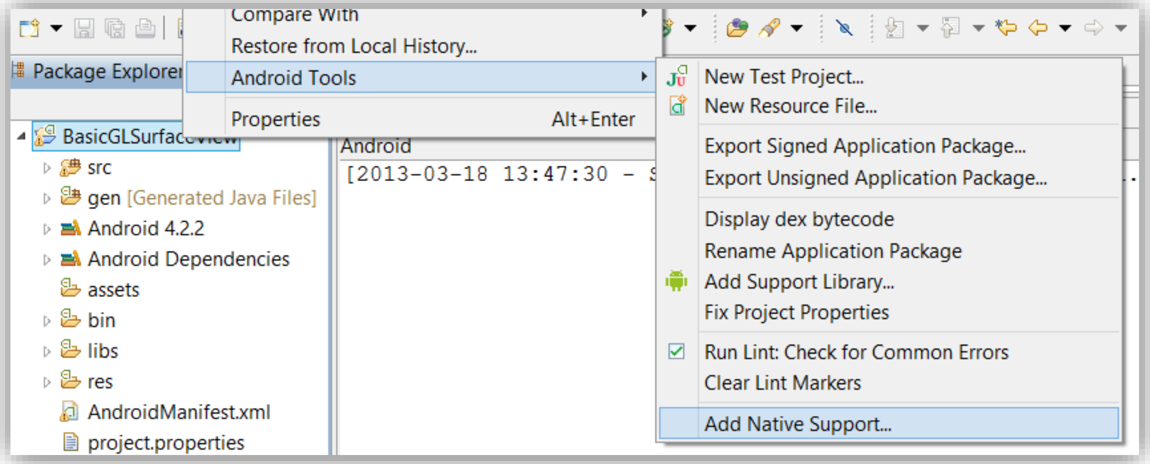
2. Reuse or create native c/c++ sources

3. Create Android.mk Makefile

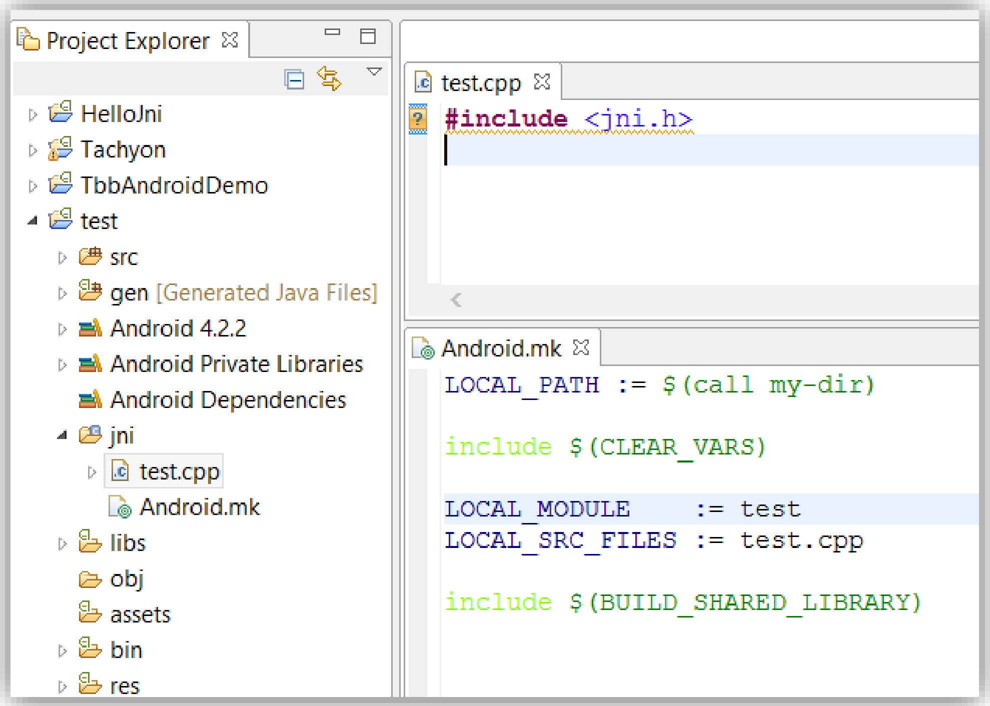
NDK-BUILD will automatically create ABI libs folders.

4. Build Native libraries using NDK-BUILD script.

Adding NDK Support to your Eclipse Android* Project



```
[2013-03-18 13:47:30 - S
```



```
#include <jni.h>
```

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE     := test
LOCAL_SRC_FILES := test.cpp

include $(BUILD_SHARED_LIBRARY)
```

Android* NDK Samples

Sample App	Type
hello-jni	Call a native function written in C from Java*.
bitmap-plasma	Access an Android* Bitmap object from C.
san-angeles	EGL and OpenGL* ES code in C.
hello-gl2	EGL setup in Java and OpenGL ES code in C.
native-activity	C only OpenGL sample (no Java, uses the NativeActivity class).
native-plasma	C only OpenGL sample (also uses the NativeActivity class).
…	

Focus on Native Activity

Only native code *in the project*

android_main() entry point running in its own thread

Event loop to get input data and frame drawing messages

```
/**
 * This is the main entry point of a native application that is using
 * android_native_app_glue.  It runs in its own thread, with its own
 * event loop for receiving input events and doing other things.
 */
void android_main(struct android_app* state);
```

Integrating Native Functions with Java*

Declare native methods in your Android* application (Java*) using the ‘native’ keyword:

```
public native String stringFromJNI();
```

Provide a native shared library built with the NDK that contains the methods used by your application:

```
libMyLib.so
```

Your application must load the shared library (before use… during class load for example):

```
static {
    System.loadLibrary("MyLib");
}
```

There is two ways to associate your native code to the Java methods: javah and JNI_OnLoad

Javah Method

“javah” helps automatically generate the appropriate JNI header stubs based on the Java* source files from the compiled Java classes files:

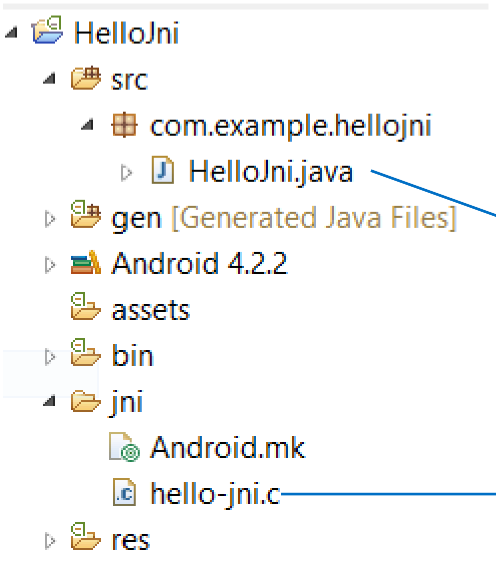
Example:

```
> javah -d jni -classpath bin/classes \

com.example.hellojni.HelloJni
```

Generates `com_example_hellojni_HelloJni.h` file with this definition:

```
JNIEXPORT jstring JNICALL
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv *, jobject);
```

Javah Method



```
HelloJni
  src
    com.example.hellojni
      HelloJni.java
  gen [Generated Java Files]
  Android 4.2.2
  assets
  bin
  jni
    Android.mk
    hello-jni.c
  res
```

```
...
    {
        ...
        tv.setText( stringFromJNI() );
        ...
    }

    public native String  stringFromJNI();

    static {
        System.loadLibrary("hello-jni");
    }
```

C function that will be automatically mapped:

```
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv* env,
        jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

JNI_OnLoad Method – Why ?

Proven method

No more surprises after methods registration

Less error prone when refactoring

Add/remove native functions easily

No symbol table issue when mixing C/C++ code

Best spot to cache Java* class object references

JNI_OnLoad Method

In your library name your functions as you wish and declare the mapping with JVM methods:

```
jstring stringFromJNI(JNIEnv* env, jobject thiz)

{ return env->NewStringUTF("Hello from JNI !");}

static JNINativeMethod exposedMethods[] = {

{"stringFromJNI","()Ljava/lang/String;",(void*)stringFromJNI},

}
```

()Ljava/lang/String; is the JNI signature of the Java* method, you can retrieve it using the javap utility:

```
> javap -s -classpath bin\classes -p com.example.hellojni.HelloJni

Compiled from "HelloJni.java“

…

public native java.lang.String stringFromJNI();

  Signature: ()Ljava/lang/String;

…
```

JNI_OnLoad Method

```
extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env;
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) !=
JNI_OK)
      return JNI_ERR;

    jclass clazz = env->FindClass("com/example/hellojni/HelloJni");
    if(clazz==NULL)
        return JNI_ERR;

    env->RegisterNatives(clazz, exposedMethods,
sizeof(exposedMethods)/sizeof(JNINativeMethod));
    env->DeleteLocalRef(clazz);

    return JNI_VERSION_1_6;
}
```

JNI_OnLoad is the library entry point called during load.

Here it applies the mapping previously defined.

Memory Handling of Java* Objects

Memory handling of Java* objects is done by the JVM:

- You only deal with references to these objects
- Each time you get a reference, you must not forget to delete it after use
- local references are automatically deleted when the native call returns to Java
- References are local by default
- Global references are only created by NewGlobalRef()

Creating a Java* String

```
C:
jstring string =
        (*env)->NewStringUTF(env, "new Java String");

C++:
jstring string = env->NewStringUTF("new Java String");
```

Memory is handled by the JVM, jstring is always a reference.

You can call DeleteLocalRef() on it once you finished with it.

Main difference with compiling JNI code in C or in C++ is the nature of “env” as you can see it here.

Remember that otherwise, the API is the same.

Getting a C/C++ String from Java* String

```
const char *nativeString = (*env)-
>GetStringUTFChars(javaString, null);
…
(*env)->ReleaseStringUTFChars(env, javaString, nativeString);

//more secure
int tmpjstrlen = env->GetStringUTFLength(tmpjstr);
char* fname = new char[tmpjstrlen + 1];
env->GetStringUTFRegion(tmpjstr, 0, tmpjstrlen, fname);
fname[tmpjstrlen] = 0;
…
delete fname;
```

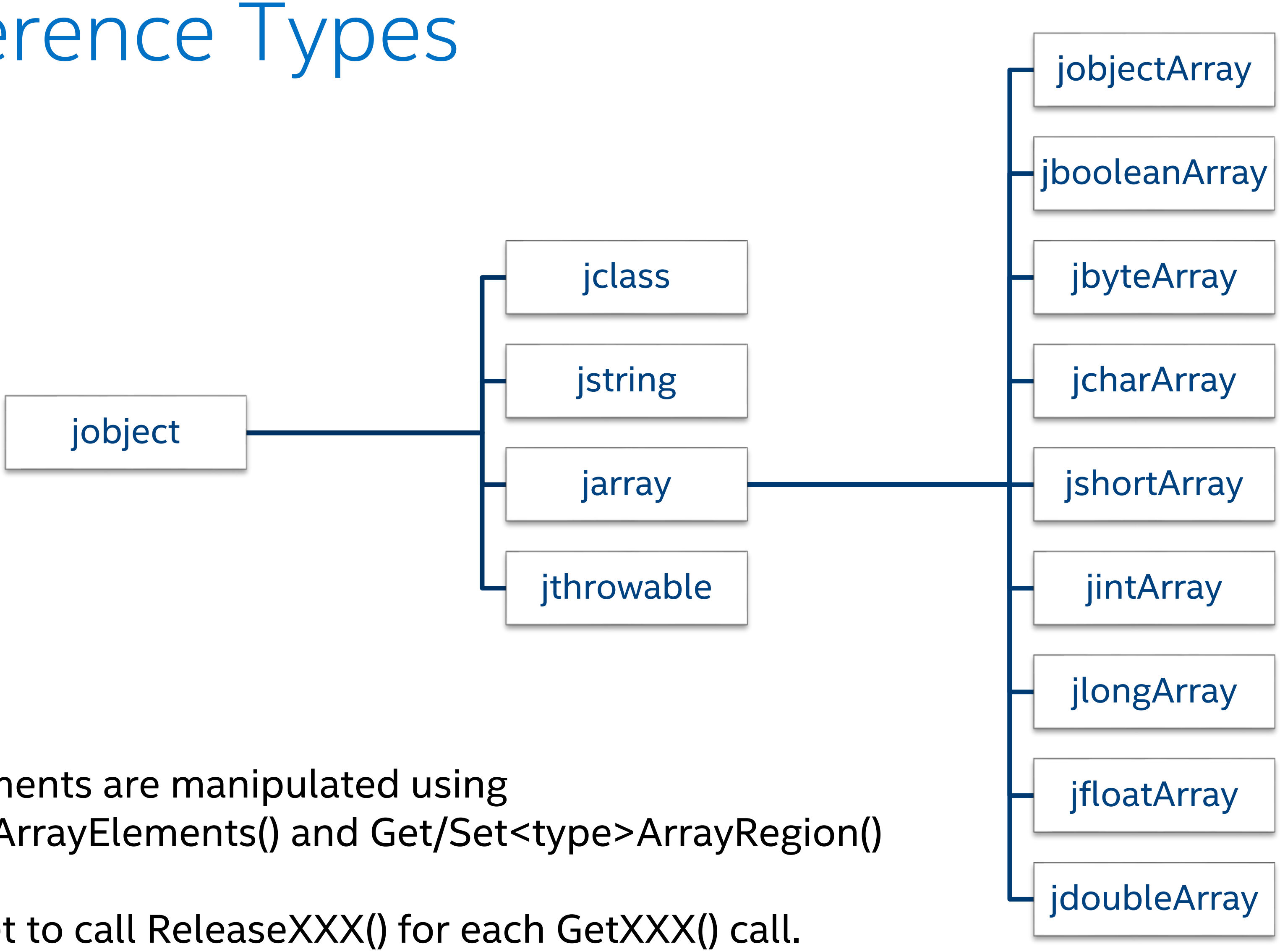
Handling Java* Exceptions

```
// call to java methods may throw Java exceptions

jthrowable ex = (*env)->ExceptionOccurred(env);
if (ex!=NULL) {
      (*env)->ExceptionClear(env);
      // deal with exception
}

(*env)->DeleteLocalRef(env, ex);
```

JNI Primitive Types

Java* Type	Native Type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	N/A

JNI Reference Types



Arrays elements are manipulated using Get<type>ArrayElements() and Get/Set<type>ArrayRegion()

Don’t forget to call ReleaseXXX() for each GetXXX() call.

Calling Java* Methods

On an object instance:

```
jclass clazz = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, clazz, "methodName",
"(…)…");
if (mid != NULL)
       (*env)->Call<Type>Method(env, obj, mid, parameters…);
```

Static call:

```
jclass clazz = (*env)->FindClass(env, "java/lang/String");
jmethodID mid = (*env)->GetStaticMethodID(env, clazz, "methodName",
"(…)…");
if (mid != NULL)
       (*env)->CallStatic<Type>Method(env, clazz, mid, parameters…);
```

- (…)…: method signature
- parameters: list of parameters expected by the Java* method
- <Type>: Java method return type

Throwing Java* Exceptions

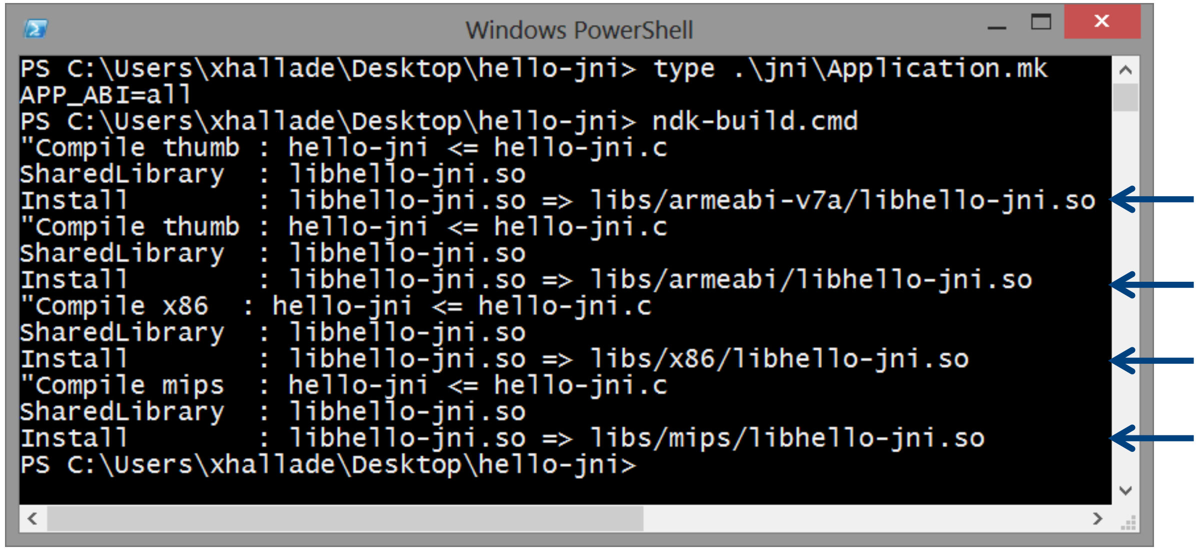
```
jclass clazz =
(*env->FindClass(env, "java/lang/Exception");

if (clazz!=NULL)
  (*env)->ThrowNew(env, clazz, "Message");
```

The exception will be thrown only when the JNI call returns to Java*, it will not break the current native code execution.

Configuring NDK Target ABIs

Include all ABIs by setting APP_ABI to all in jni/**Application.mk**:

```
APP_ABI=all
```



```
PS C:\Users\xhallade\Desktop\hello-jni> type .\jni\Application.mk
APP_ABI=all
PS C:\Users\xhallade\Desktop\hello-jni> ndk-build.cmd
"Compile thumb : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi-v7a/libhello-jni.so
"Compile thumb : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
"Compile x86  : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/x86/libhello-jni.so
"Compile mips  : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/mips/libhello-jni.so
PS C:\Users\xhallade\Desktop\hello-jni>
```

Build ARM v7a libs

Build ARM v5 libs

Build x86 libs

Build mips libs

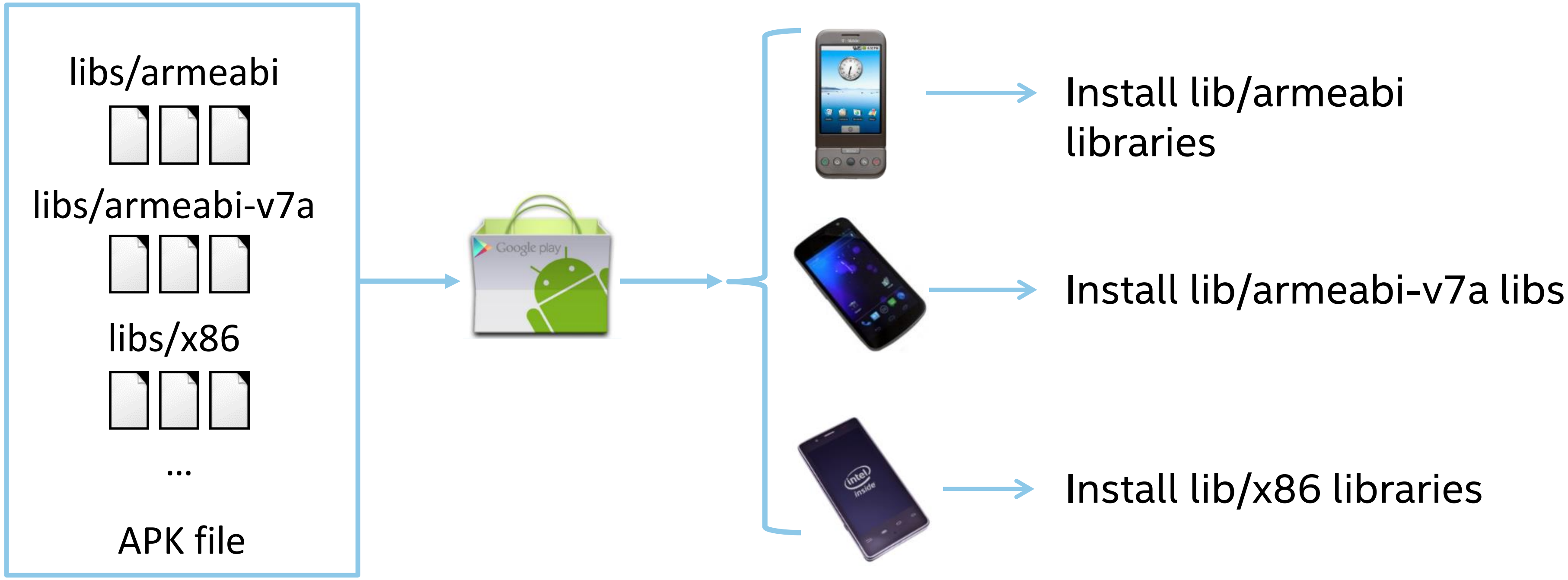
The NDK will generate optimized code for all target ABIs

You can also pass APP_ABI variable to ndk-build, and specify each ABI:

```
ndk-build APP_ABI=x86
```

“Fat” APKs

By default, an APK contains libraries for every supported ABIs.



libs/armeabi

libs/armeabi-v7a

libs/x86

…

APK file

Install lib/armeabi libraries

Install lib/armeabi-v7a libs

Install lib/x86 libraries

Libs for the selected ABI are installed, the others remain inside the downloaded APK

Multiple APKs

Google Play* supports multiple APKs for the same application.

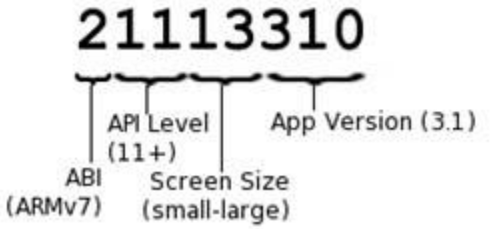
What compatible APK will be chosen for a device entirely depends on the android:versionCode

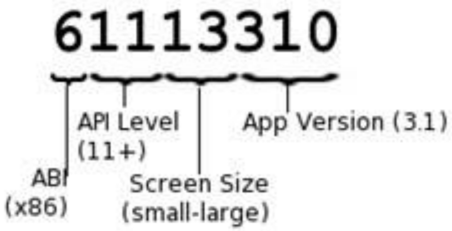
If you have multiple APKs for multiple ABIs, best is to simply prefix your current version code with a digit representing the ABI:

2310 ↓ **ARMv7**

6310 ↓ **x86**

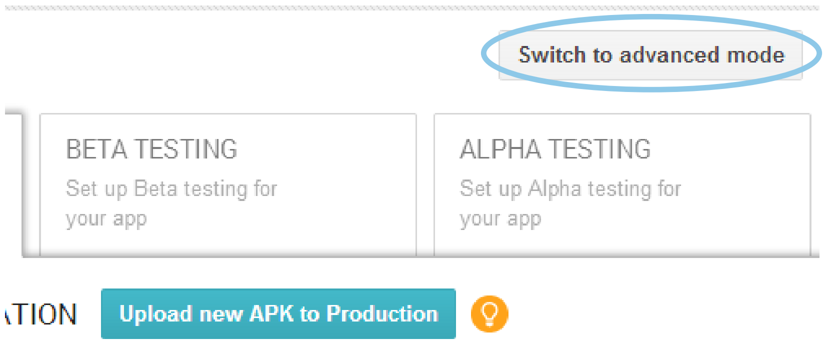
You can have more options for multiple APKs, here is a convention that will work if you're using all of these:





Uploading Multiple APKs to the store

Switch to Advanced mode before uploading the second APK.

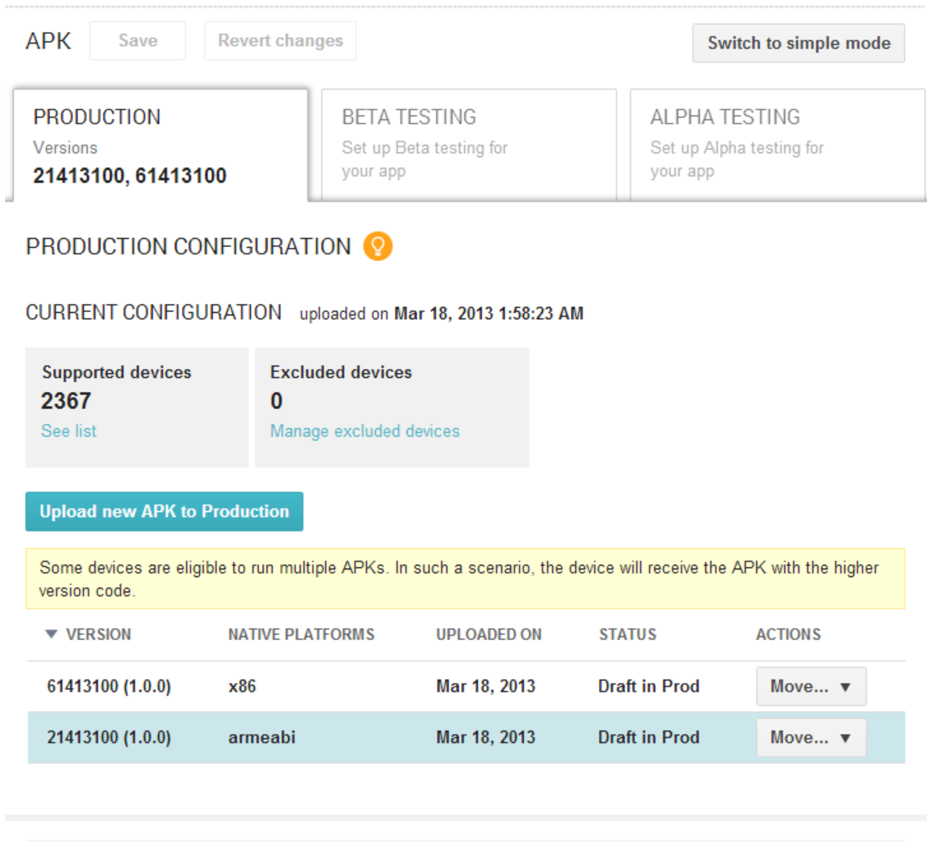


Switch to advanced mode

BETA TESTING
Set up Beta testing for your app

ALPHA TESTING
Set up Alpha testing for your app

TION Upload new APK to Production



APK Save Revert changes Switch to simple mode

PRODUCTION
Versions
21413100, 61413100

BETA TESTING
Set up Beta testing for your app

ALPHA TESTING
Set up Alpha testing for your app

PRODUCTION CONFIGURATION

CURRENT CONFIGURATION uploaded on **Mar 18, 2013 1:58:23 AM**

Supported devices
2367
See list

Excluded devices
0
Manage excluded devices

Upload new APK to Production

Some devices are eligible to run multiple APKs. In such a scenario, the device will receive the APK with the higher version code.

▼ VERSION	NATIVE PLATFORMS	UPLOADED ON	STATUS	ACTIONS
61413100 (1.0.0)	x86	Mar 18, 2013	Draft in Prod	Move... ▼
21413100 (1.0.0)	armeabi	Mar 18, 2013	Draft in Prod	Move... ▼

Debugging with logcat

NDK provides log API in <android/log.h>:

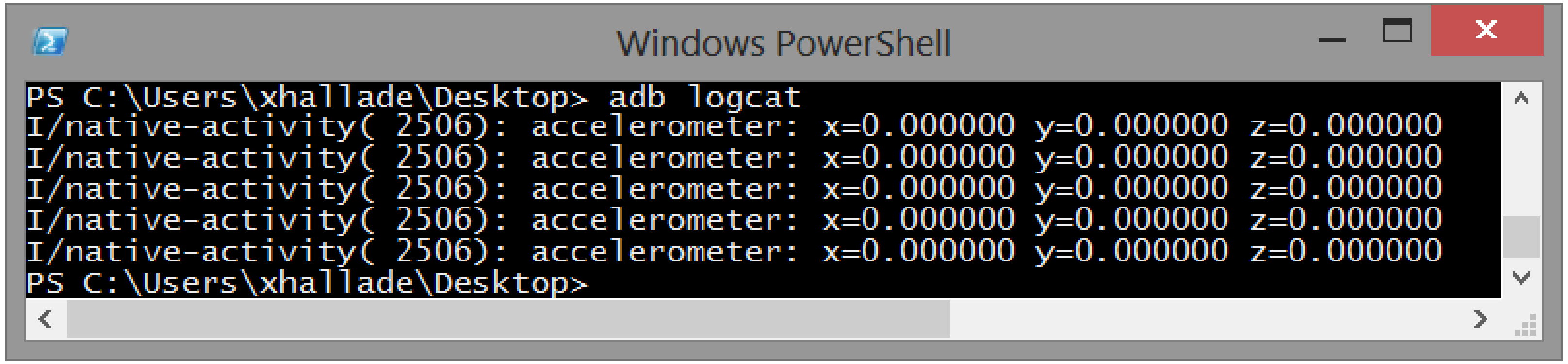
```
int __android_log_print(int prio, const char *tag,
     const char *fmt, ...)
```

Usually used through this sort of macro:

```
#define LOGI(...) ((void)__android_log_print(ANDROID_LOG_INFO, "APPTAG", __VA_ARGS__))
```

Usage Example:

LOGI("accelerometer: x=%f y=%f z=%f", x, y, z);



```
PS C:\Users\xhallade\Desktop> adb logcat
I/native-activity( 2506): accelerometer: x=0.000000 y=0.000000 z=0.000000
I/native-activity( 2506): accelerometer: x=0.000000 y=0.000000 z=0.000000
I/native-activity( 2506): accelerometer: x=0.000000 y=0.000000 z=0.000000
I/native-activity( 2506): accelerometer: x=0.000000 y=0.000000 z=0.000000
I/native-activity( 2506): accelerometer: x=0.000000 y=0.000000 z=0.000000
PS C:\Users\xhallade\Desktop>
```

Debugging with logcat

To get more information on native code execution:

```
adb shell setprop debug.checkjni 1
```

(already enabled by default in the emulator)

And to get memory debug information (root only):

```
adb shell setprop libc.debug.malloc 1
```

-> leak detection

```
adb shell setprop libc.debug.malloc 10
```

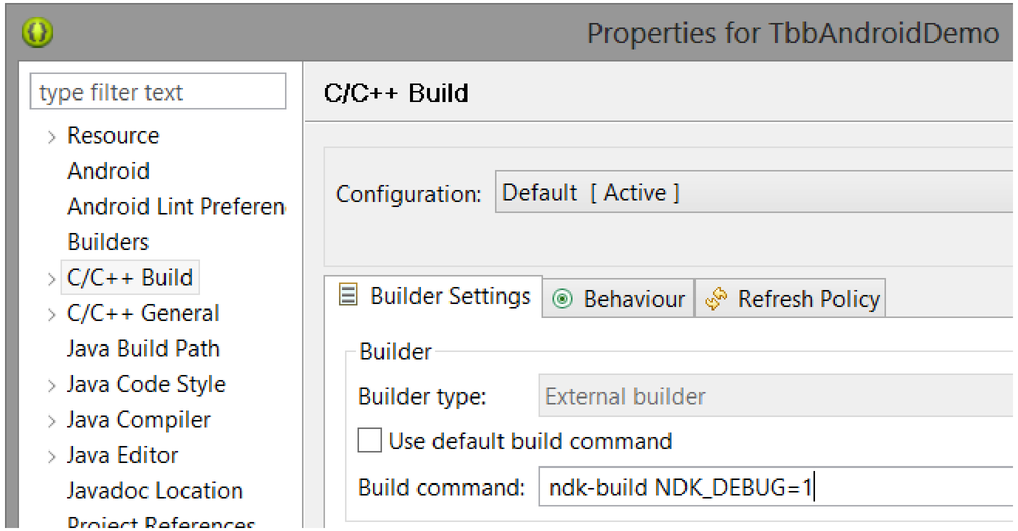
-> overruns detection

`adb shell start/stop` -> reload environment

Debugging with GDB and Eclipse

Native support must be added to your project

Pass NDK_DEBUG=1 APP_OPTIM=debug to the ndk-build command, from the project properties:



Properties for TbbAndroidDemo

type filter text

- Resource
- Android
- Android Lint Preferen
- Builders
- C/C++ Build
- C/C++ General
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project References

C/C++ Build

Configuration: Default [Active]

Builder Settings | Behaviour | Refresh Policy

Builder

Builder type: External builder

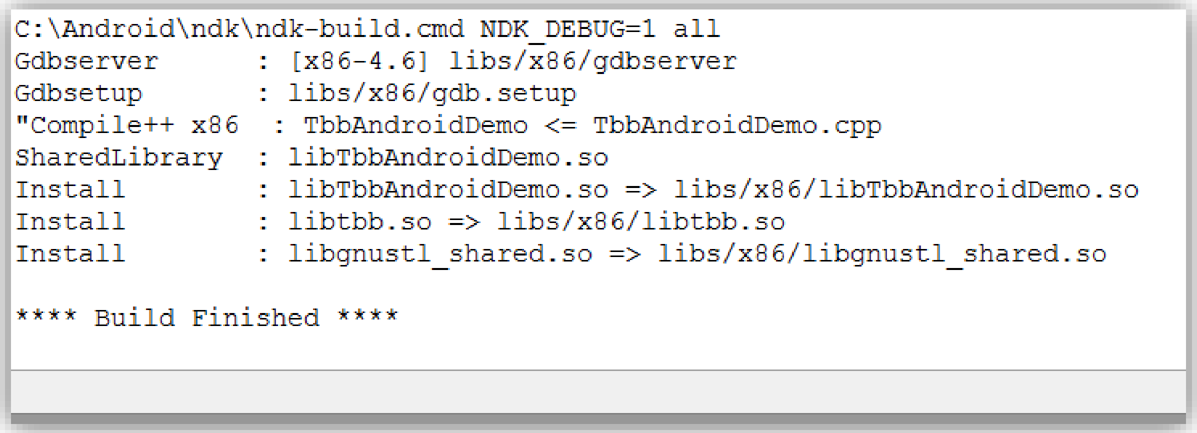
□ Use default build command

Build command: ndk-build NDK_DEBUG=1

NDK_DEBUG flag is supposed to be automatically set for a debug build, but this is not currently the case.

Debugging with GDB and Eclipse*

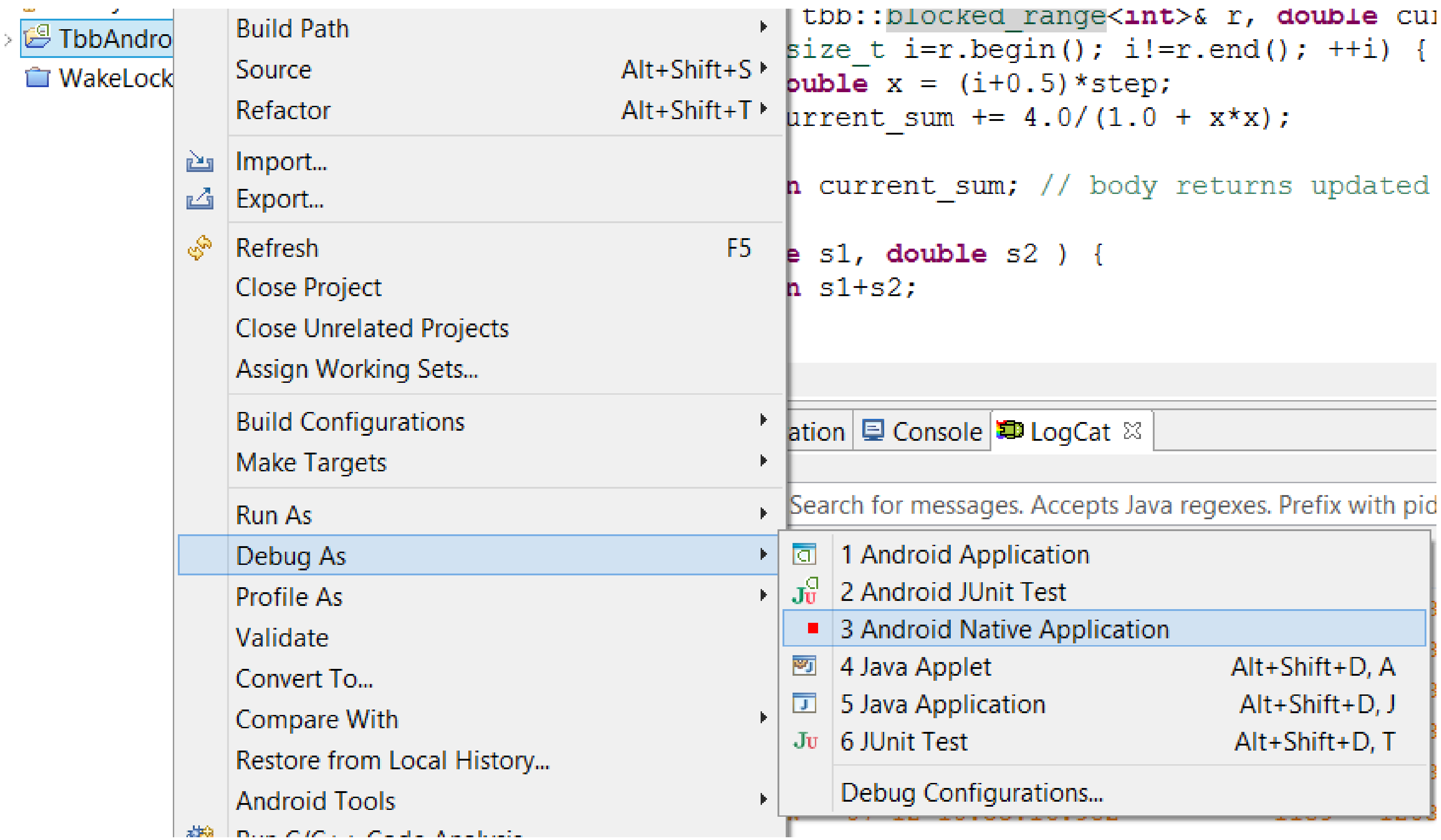
When NDK_DEBUG=1 is specified, a “gdbserver” file is added to your libraries



```
C:\Android\ndk\ndk-build.cmd NDK_DEBUG=1 all
Gdbserver      : [x86-4.6] libs/x86/gdbserver
Gdbsetup       : libs/x86/gdb.setup
"Compile++ x86  : TbbAndroidDemo <= TbbAndroidDemo.cpp
SharedLibrary  : libTbbAndroidDemo.so
Install        : libTbbAndroidDemo.so => libs/x86/libTbbAndroidDemo.so
Install        : libtbb.so => libs/x86/libtbb.so
Install        : libgnustl_shared.so => libs/x86/libgnustl_shared.so

**** Build Finished ****
```

Debugging with GDB and Eclipse*

Debug your project as a native Android* application:

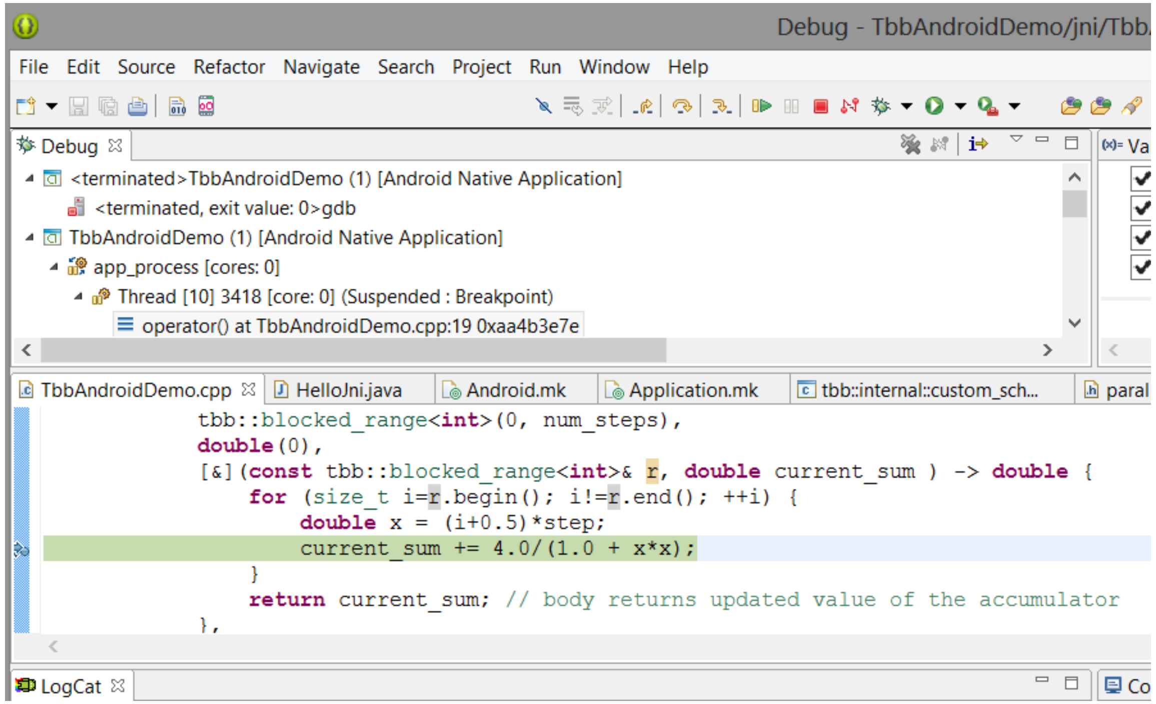


```
 tbb::blocked_range<int>& r, double cu
size_t i=r.begin(); i!=r.end(); ++i) {
ouble x = (i+0.5)*step;
urrent_sum += 4.0/(1.0 + x*x);

n current_sum; // body returns updated

e s1, double s2 ) {
n s1+s2;
```

Search for messages. Accepts Java regexes. Prefix with pid

Debugging with GDB and Eclipse

From Eclipse “Debug” perspective, you can manipulate breakpoints and debug your project



```
<terminated>TbbAndroidDemo (1) [Android Native Application]
    <terminated, exit value: 0>gdb
TbbAndroidDemo (1) [Android Native Application]
    app_process [cores: 0]
        Thread [10] 3418 [core: 0] (Suspended : Breakpoint)
            operator() at TbbAndroidDemo.cpp:19 0xaa4b3e7e
```

```
tbb::blocked_range<int>(0, num_steps),
double(0),
[&](const tbb::blocked_range<int>& r, double current_sum ) -> double {
    for (size_t i=r.begin(); i!=r.end(); ++i) {
        double x = (i+0.5)*step;
        current_sum += 4.0/(1.0 + x*x);
    }
    return current_sum; // body returns updated value of the accumulator
},
```

Your application will run before the debugger is attached, hence breakpoints you set near application launch will be ignored

GCC Flags

```
ifeq ($(TARGET_ARCH_ABI),x86)
LOCAL_CFLAGS += -ffast-math -mtune=atom -mssse3 -mfpmath=sse
else
LOCAL_CFLAGS += ...
endif
```

To optimize for Intel Silvermont Microarchitecture (available starting with NDK r9 gcc-4.8 toolchain):

```
LOCAL_CFLAGS += -O3 -ffast-math -mtune=slm -msse4.2 -mfpmath=sse
```

ffast-math influence round-off of fp arithmetic and so breaks strict IEEE compliance

The other optimizations are totally safe

Add -ftree-vectorizer-verbose to get a vectorization report

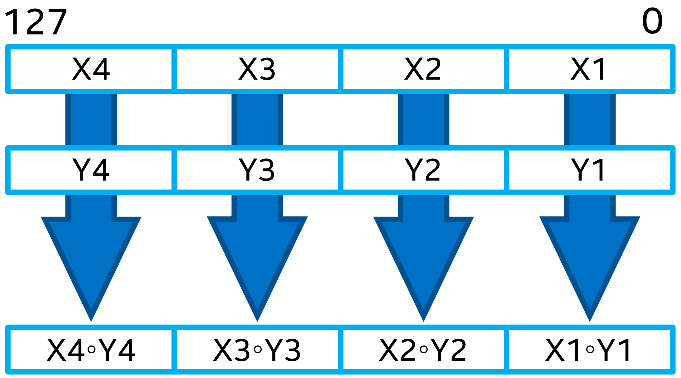
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Vectorization

SIMD instructions up to SSSE3 available on current Intel® Atom™ processor based platforms, Intel® SSE4.2 on the Intel Silvermont Microarchitecture



127 0

X4 X3 X2 X1

Y4 Y3 Y2 Y1

X4◦Y4 X3◦Y3 X2◦Y2 X1◦Y1

SSSE3, SSE4.2

Vector size: **128 bit**

Data types:

- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 2, 4, 8, 16

On ARM*, you can get vectorization through the ARM NEON* instructions

Two classic ways to use these instructions:

- Compiler auto-vectorization
- Compiler intrinsics

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Android* Studio NDK support

- Having .c(pp) sources inside jni folder ?
  - ndk-build automatically called on a generated Android.mk, ignoring any existing .mk
  - All configuration done through build.gradle (moduleName, ldLibs, cFlags, stl)
  - You can change that to continue relying on your own Makefiles:

    http://ph0b.com/android-studio-gradle-and-ndk-integration/

- Having .so files to integrate ?
  - Copy them to jniLibs folder or integrate them from a .aar library
- Use flavors to build one APK per architecture with a computed versionCode

  http://tools.android.com/tech-docs/new-build-system/tips

Q&A

xavier.hallade@intel.com

@ph0b – ph0b.com